Low-power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction

Xun Jian
University of Illinois
at Urbana-Champaign
xunjian1@illinois.edu

Henry Duwe
University of Illinois
at Urbana-Champaign
duweiii2@illinois.edu

John Sartori
University of Minnesota
jsartori@umn.edu

Vilas Sridharan
AMD Research
Advanced Micro Devices, Inc.
vilas.sridharan@amd.com

Rakesh Kumar
University of Illinois
at Urbana-Champaign
rakeshk@illinois.edu

ABSTRACT

Due to their large memory capacities, many modern servers require chipkill correct, an advanced type of memory error detection and correction, to meet their reliability requirements. However, existing chipkill-correct solutions incur high power or storage overheads, or both because they use dedicated error-correction resources per codeword to perform error correction. This requires high overhead for correction and results in high overhead for error detection. We propose a novel chipkill-correct solution, multi-line error correction, that uses resources shared across multiple lines in memory for error correction to reduce the overhead of both error detection and correction. Our evaluations show that the proposed solution reduces memory power by a mean of 27%, and up to 38% with respect to commercial solutions, at a cost of 0.4% increase in storage overhead and minimal impact on reliability.

Categories and Subject Descriptors

B.7.3 [**Reliability and Testing**]: Error-checking; B.3.2 [**Memory Structures**]: Primary Memory

1. INTRODUCTION

As the memory capacity of servers increases, memory power consumption and reliability are becoming increasingly serious concerns. For example, memory power already consumes 25% to 40% of total server power [25]. With increasingly large memory systems, memory errors such as detectable uncorrectable errors (DUEs) and silent data corruption (SDC) will grow increasingly frequent if strong error detection and correction are not deployed in memory. Memory errors not only incur high cost in terms of server downtime (e.g., a DUE often causes a system crash [21]), but can also result in unrecoverable failures (e.g., failures due to SDCs) that can be even more costly. However, implementing strong error detection and correction in memory to help modern servers meet their reliability requirements typically requires overheads that further increase memory power consumption. Providing strong error detection and correction at low memory power consumption, therefore, becomes an important goal to achieve.

Currently, many servers employ chipkill correct to meet the desired reliability [1, 11]. Chipkill correct is an advanced type of memory-error correction that significantly improves memory reliability compared to the well-known single-error correction/double-error detection (SECDED) by providing correct memory accesses even when a DRAM device has failed completely [23]. Existing large-scale systems that use chipkill correct include Google server farms [21] and numerous supercomputers [23, 10].

In a typical chipkill-correct memory system, each word in memory is stored in the form of a codeword, which is the data word plus redundant check bits. Each codeword is typically broken down into groups of bits, called *symbols*, which are evenly striped across multiple DRAM devices. This group of DRAM devices form a *rank*. Existing chipkill-correct solutions correct one bad symbol per codeword and detect two bad symbols per codeword; this is achieved by using a single-symbol correct/double-symbol detect (SSCDSD) linear block code [26, 4, 1, 11].

Unfortunately, SSCDSD codes incur high overhead because they require a minimum of three check symbols per codeword [26, 4]. Because check symbols are redundant symbols that do not hold actual data values, such symbols result in a storage overhead. To keep the overall storage overhead low, commercial chipkill-correct solutions amortize the overhead of the check symbols over a large number of data symbols per codeword. They use 32 data symbols and four check symbols per codeword, which results in ranks with 32 data devices and four redundant devices with a storage overhead of $4/32 = 12.5\%$ [26, 4]. Having 36 devices per rank leads to high memory power consumption because every device in a rank must be accessed per memory request [26, 4]. In comparison, SECDED typically employs only nine devices per rank, consisting of eight data devices and one redundant device, which is only one-fourth the number of devices required by commercial chipkill-correct solutions. A memory system with only nine devices per rank has been reported to reduce

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC 13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503243

power by up to 45% compared to a memory system with 36 devices per rank [24].

Unlike the commercial chipkill-correct solution just described which optimizes for low storage overhead, recent work on chipkill correct proposes reducing memory power consumption at the expense of increased storage overhead by reducing the number of data symbols per codeword [4, 26]. However, increasing storage overhead increases the cost of the memory system as well as background power consumption because more physical memory is required to provide the same amount of usable memory.

Instead of simply reducing the number of data symbols per codeword, we explore reducing the number of check symbols per codeword (e.g., down to one) while maintaining similar DUE and SDC rates. This allows the rank size to be reduced without having to increase storage overhead.

First, we observe that the high overheads in existing chipkill correct solutions are due to using a dedicated correction-check symbol per codeword to provide error correction. Using a dedicated correction-check symbol for error correction unfortunately reduces the effectiveness of error detection, so more check symbols for detection are needed to meet a desired error-detection strength. Second, we observe that using dedicated check symbols for error correction also results in unnecessarily high overhead for error correction. Recent field studies of DRAM faults show that the vast majority of errors in memory are caused by device-level faults, which are correlated faults within a single device [23, 10]. As such, errors in physically adjacent codewords in a chipkill-correct memory system typically lie in the same symbol position. Therefore, each codeword does not need to have its own dedicated correction-check symbol because a primary function of the correction-check symbol is to locate the symbol in the codeword that contains error. Instead, it suffices to let multiple adjacent codewords share a common set of error localization resources with only minimal impact on DUE rate.

Based on these observations, we propose avoiding the high overhead of error detection by replacing the error-correction method used in prior chipkill-correct solutions (i.e., error correction via a dedicated correction-check symbol per codeword) with erasure correction. Erasure correction is a method to perform error correction in linear block codes when bad symbols have been localized using some other means (e.g., using checksums). Using erasure correction for error correction reduces the number of check symbols required per codeword to one. However, providing erasure correction on the conventional codeword-by-codeword basis may incur unacceptably high storage overhead because this requires each symbol position in a codeword have its own error-localization checksums. Therefore, we also propose providing error correction at a coarse granularity (i.e., over groups of lines in memory) to reduce storage overhead. Our evaluations using SPEC and PARSEC workloads show that compared to commercial chipkill-correct solutions, our proposed chipkill-correct solution, multi-line error correction (Multi-ECC), reduces memory power by a mean of 27% and up to 38% with only 0.4% higher storage overhead while providing similar DUE and SDC rates.

2. RELATED WORK

Commercial chipkill-correct solutions correct a single bad symbol and detect up to two bad symbols in each codeword [4, 26, 24, 4]. A symbol is simply a group of adjacent bits; a symbol is referred to as a data symbol, if it holds application data, or as a check symbol, if it holds the redundant information for error detection or correction. A group of data symbols and the check symbols protecting that group of data symbols together form a codeword. Figure 1 shows a memory rank protected by a commercial chipkill-correct solution. In the figure, every symbol of a codeword is stored in a different device in the rank. As a result, even in the event of a complete device failure, only a single symbol in each codeword is lost as long as other devices are fault-free; the corrupted symbol can be detected and then reconstructed using the remaining good symbols in the codeword.

Commercial chipkill-correct solutions require a minimum of three check symbols per codeword to provide SSCDSD [26, 4]. In practice, however, commercial chipkill-correct solutions use four check symbols per codeword to abide by established ECC memory-processor interfaces.

To keep the storage overhead low despite the overhead of four check symbols per codeword, commercial chipkill-correct solutions assign 32 data symbols per codeword; this results in a storage overhead of $4/32 = 12.5\%$, which is the same as SECDED. Because each codeword is striped across 32 data devices and four redundant devices, as shown in Figure 1, all 36 devices must be accessed for each memory request, which leads to high memory power consumption.

Several recent proposals aim to reduce the power consumption of commercial chipkill-correct memory systems. Ahn et al. [4] propose trading storage overhead for memory power consumption by reducing the number of data devices per rank. However, this requires design of custom DIMMs and memory-processor interfaces, which is expensive. VECC [26] avoids custom DIMMs by remapping error correction check symbols to data devices via virtualization. VECC uses 16 data devices and two redundant devices per rank to store the 16 data symbols and two check symbols per codeword, and remaps the third check symbol, used for error correction, to the data region of memory in a different rank. Because two out of the 18 devices per rank are used to store the detection symbols, and because there is one correction-check symbol for every 16 data symbols in the data region of memory, the fraction of total memory that can be used by an application to store data is $dataFrac = (16/18) \cdot (15/16)$. The storage overhead is, therefore, $(1 - dataFrac)/(dataFrac) = 21.9\%$, instead of the conventional 12.5%.

Another recent proposal to reduce the overhead of chipkill correct is LOT-ECC, which moves away from linear block codes to avoid the high overhead of multiple check symbols per codeword [24]. By moving away from linear block codes,

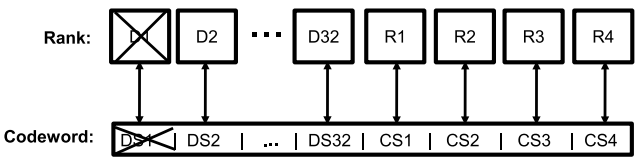

Figure 1: Data layout of commercial chipkill-correct solutions. *D* stands for data device, *DS* stands for data symbol, *R* stands for redundant device, and *CS* stands for check symbol.

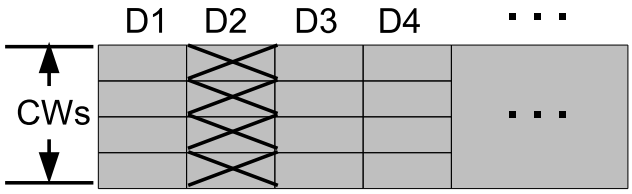

Figure 2: Error manifestation of a device-level fault that affects a group of codewords in adjacent rows that share the same column address. *D* stands for device, *CW* for a codeword, and *X* for a faulty symbol in a codeword.

LOT-ECC reduces the number of devices per rank to nine. However, LOT-ECC suffers from increased storage overhead (e.g., 26.5% instead of 12.5% [24]). Also, LOT-ECC suffers from significantly lower error-detection coverage than prior chipkill-correct solutions due to moving away from linear block code. To perform error detection, LOT-ECC computes error-detection bits independently for each device and stores them in the same location in the same device as the data they protect. As such, a fault in the address decoder of a device may lead to both data and error-detection bits being accessed from a wrong location in the device, resulting in undetectable error even if all other devices in the rank are completely fault-free. In comparison, three devices have to develop faults in the same memory location to result in undetectable error in commercial chipkill-correct solutions. Note that address-decoder faults may be a common DRAM fault mode in the field [22].

3. MOTIVATION

Existing linear block code-based chipkill-correct solutions use a dedicated check symbol per codeword for error correction. The benefit of using a dedicated error correction-check symbol per codeword for error correction is allowing a bad symbol in each codeword to be corrected, independent of bad symbols in other codewords. This error-correction capability is desirable for scenarios when errors tend to appear in different symbols in different codewords. However, we observe that errors tend to appear in the same symbol in different codewords in a chipkill-correct memory system. Because each device always contributes the same symbol to every codeword in a chipkill-correct memory system, a single faulty device always causes the same symbol – not different symbols – in the codewords to become erroneous. Figure 2 illustrates how a device-level fault affecting a single device per rank, such as the column, bank, device, or lane fault (which have been reported to cause the vast majority of errors in memory [23, 10]), affects a group of codewords in adjacent rows that share the same column address. Based on the preceding observation, letting multiple codewords share the same set of error-correction resources – and thereby relaxing the error correction capability of chipkill correct such that errors in a group of codewords can be corrected only if every error is located in the same symbol position – one can reduce the storage overhead of error correction with only minimal impact on correction coverage.

Meanwhile, relying on dedicated check symbol(s) per codeword to perform error correction also increases the overhead of error detection. This is because error correction via dedicated correction-check symbols is performed through maximum-likelihood decoding, whereby a received codeword is decoded as the valid codeword that is closest to the received codeword in terms of the Hamming distance [9]. The Hamming distance between two codewords is defined as the number of symbols that are different between the codewords; for example, with r check symbols per codeword, the Hamming distance between two neighboring valid codewords (i.e., codewords that differ in only one data symbol) is $1 + r$. Due to maximum-likelihood decoding, undetectable errors occur whenever the received codeword is closer to another valid codeword than the intended codeword (i.e., the correct codeword that was originally written to memory).

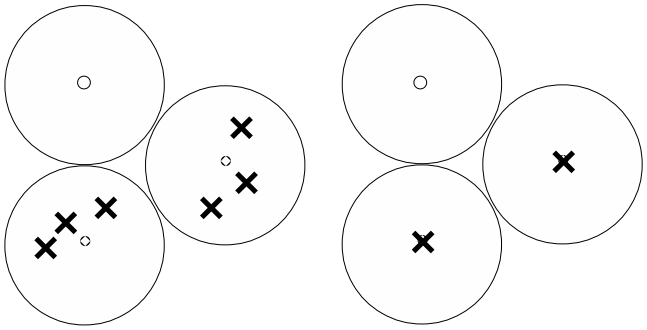

Figure 3: Received codewords that result in SDC in the presence of (left) and absence of (right) of maximum-likelihood decoding. The hollow dots represent intended codewords. The solid dots represent other valid codewords. The crosses represent received codewords that result in SDCs.

On the other hand, in the absence of maximum-likelihood decoding (i.e., the codewords do not contain dedicated check symbols for error correction), undetectable errors occur only when the received codeword is the same as a valid codeword that is different from the intended codeword. Figure 3 illustrates the effect of maximum-likelihood decoding on error detection. The left half of the figure shows that when maximum-likelihood decoding is used, any received codeword (represented by a cross) that is closer to a neighboring valid codeword (represented by a solid dot) than the intended codeword (represented by the hollow dot) leads to undetectable error. On the other hand, in the absence of maximum-likelihood decoding, undetected error occurs only when the received codeword exactly matches a valid codeword that is different from the intended codeword; this is illustrated in the right half of Figure 3 by the crosses lying on top of the solid dots.

To illustrate how the fundamental difference in error detection capabilities of these two scenarios affect the overhead of error detection, consider the following examples. When there is only a single check symbol per codeword, the check symbol can detect all single-symbol errors and provide high detection coverage for double-symbol errors. When a correction-check symbol is added to the codeword for error correction, the first check symbol can no longer detect any double-symbol errors. This is because when double-symbol error occurs, maximum-likelihood decoding estimates that the intended codeword should be only a Hamming distance of two away from the received codeword, when the received

codeword with double-symbol error is actually a Hamming distance of $1+2 = 3$ away from the intended codeword. This results in wrong correction and, therefore, SDC. To detect double-symbol errors, a third check symbol must be added to the codeword. Thus, prior chipkill-correct solutions that use a dedicated correction check-symbol for error correction require a minimum of three check symbols per codeword. Conversely, the overhead of error detection can be greatly reduced if correction of a codeword can be provided by some other correction mechanism (e.g., erasure correction) that does not involve the the use of dedicated correction-check symbols.

4. MULTI-ECC

Unlike prior chipkill-correct solutions that use a dedicated correction-check symbol per codeword for error correction, Multi-ECC uses erasure correction. This allows Multi-ECC to avoid maximum-likelihood correction and, thereby, significantly reduce error-detection overhead compared to prior solutions. Under Multi-ECC, each codeword needs a single check symbol for both error detection and erasure correction. As a result, only a single redundant device is required per rank.

To implement the error localization step required for erasure correction, Multi-ECC uses checksums stored in each device in the rank, including the data devices. However, because the checksums would incur unacceptably high storage overhead if they were to be allocated to every codeword (e.g., 50% because erasure correction requires a checksum corresponding to each symbol position in a codeword), Multi-ECC shares a common set of checksums across a group of codewords. Erasure correction allows Multi-ECC to correct single-symbol errors and detect double-symbol errors while using only nine devices per rank (eight data devices + one redundant device) to reduce dynamic memory power consumption significantly compared to prior chipkill-correct solutions. Section 4.1 and 4.3 describe error detection and correction in detail.

4.1 Error Detection

Multi-ECC uses a linear block code with a single check symbol for error detection but does not use any dedicated check symbols per codeword for error correction to avoid maximum-likelihood correction. As such, the one check symbol guarantees detection of all single-symbol errors and provides high detection coverage of multi-symbol errors. Because Multi-ECC uses only a single check symbol per codeword, it requires only nine devices per rank compared to 36 in commercial chipkill-correct solutions and 18 in VECC.

The specific type of linear block code used by Multi-ECC is the Reed-Solomon (RS) code, which is also used by VECC and many existing chipkill-correct solutions [26, 4]. However, unlike previous solutions that use 8-bit and 4-bit RS codes (e.g., each symbol consists of eight or four bits) [26, 4], Multi-ECC uses a 16-bit RS code to provide high detection coverage of double-symbol errors while requiring only a single check symbol per codeword.

To understand this choice of symbol width, let us first define the syndrome as the difference between the check symbol computed when one of the symbols is bad and the check symbol computed when all the symbols are error-free. Because a syndrome can take any value between 0 and $2^n - 1$, where n is the number of bits per symbol, we observe that when there are two bad symbols in a codeword, there should be only one chance in 2^n that the syndrome introduced by one of the two bad symbols exactly cancels out the syndrome introduced by the other bad symbol, which results in an undetectable error. Therefore, we choose a large value of n (i.e., $n = 16$) to provide high detection coverage of double-symbol errors.

To confirm that a 16-bit RS code can indeed detect $1 - 2^{-16} \approx 99.9985\%$ of double-symbol errors, we performed Monte Carlo experiments[1] to simulate double-symbol errors in randomly generated codewords. We injected errors into two randomly selected symbols in the codeword, including the check symbol. To confirm that the RS check symbol provides high error-detection coverage for both single- and multi-bit errors, which are reported to be roughly equally common [23], our experiments include two sets of Monte Carlo simulations in which one set injects only single bit-flips per bad symbol and the other injects a random number of bit-flips per bad symbol. The simulation results show that the probability of undetectable single-symbol error is 0 and the probability of undetectable double-symbol error is less than 2^{-16}.

If one wishes to reduce further the probability of undetectable double-symbol errors, one can increase the number of bits per symbol. However, increasing the number of bits per symbol to 32 may result in unacceptably high decoder overhead (refer to Section 5.1 on decoder overhead). A second method is to double the number of symbols per codeword so that each device stores two 16-bit symbols from each codeword. This increases the number of check symbols per codeword to two without increasing the storage overhead. When there are two check symbols per codeword, both syndromes of both check symbols must be zero for an error to go undetected. Therefore, the probability that an error spanning across multiple devices goes undetected is reduced down to $(2^{-16})^2 \approx 2 \cdot 10^{-10}$. Because two check symbols guarantee detection of two bad symbols, and because each device stores only two symbols from each codeword, any error affecting a single device is still guaranteed to be detected.

4.2 Impact on SDC Rate

As described in the previous section, by using a RS code with a single check symbol and 16 bits per codeword, Multi-ECC guarantees detection of single-symbol errors and detects more than $1 - 2^{-16} = 99.9985\%$ of double-symbol errors. What does having a double-symbol error-detection coverage of more than 99.9985% mean in terms of the more standard metric of SDC rate? It implies that if double-symbol errors in a system were to occur at a rate of once per x years, then the rate of undetectable double-symbol errors is once per $x/(1 - 99.9985\%)$ years.

Unfortunately, the incidence rates of double-symbol errors in memory, which are DUEs for memory systems with chipkill correct, are not available in literature. To calculate the rate of undetectable double-symbol errors in Multi-ECC, we instead rely on a commercial target DUE rate for servers of once per 10 years per server [7]. By pessimistically equating the incidence rate of double-symbol errors in the memory subsystem with the target DUE rate of the entire system, we calculate that in the worst case the rate of undetectable double-symbol errors is at most once per

[1]We used a RS code with a primitive polynomial of $X^{16} + X^{12} + X^3 + X + 1$ and a generator polynomial of X+60000.

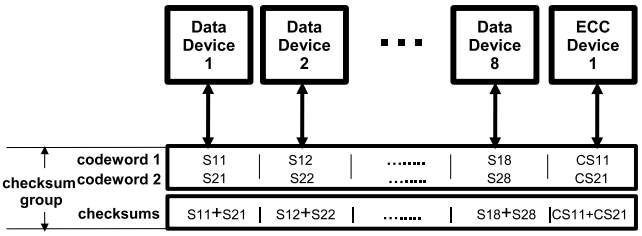

Figure 4: Example checksum. + represents the computation of checksums from data symbols, *S* is short for symbol, and *CS* stands for a RS check symbol.

$10/(1-99.9985\%) \approx 650000$ years per server. Compared to a commercial target SDC rate of one SDC per 1000 years per server [7], one undetectable double-symbol error per 650000 years per server represents only $(1/650000)/(1/1000) \approx 0.15\%$ of the target SDC rate. One can also optionally use the second technique described in Section 4.1 to achieve an undetectable double-symbol error rate of once per $10/2 \cdot 10^{-10} \approx$ 50 billion years per server. If the smaller 8-bit symbols were used instead, the rate of undetectable double-symbol errors is as high as 39% of the target SDC rate.

4.3 Error Correction

An excellent property of RS codes is that the same check symbol used for error detection can be re-used to provide *erasure* correction; that is, the bad symbol detected by a check symbol also can be corrected via the same check symbol if the exact location of the bad symbol is known [9]. To localize the bad symbol in a codeword for the purpose of erasure correction, we use intra-device checksums, which are checksums stored in each device (including the data devices), to localize the faulty device, and thereby localize the bad symbol. Multi-ECC employs one's complement checksums, used for error detection in LOT-ECC, for error localization. As explained in LOT-ECC, one's complement checksums guarantee the detection of stuck-at-1 and stuck-at-0 faults, which are common fault modes in DRAMs. Note that LOT-ECC uses the checksums for error detection, while Multi-ECC uses the checksums strictly for the error-localization step required for erasure correction.

However, being able to perform erasure correction at the granularity of every word will require unacceptably high storage overhead, because each symbol position requires a corresponding checksum to provide error localization. In Section 3, we observe that faults affecting multiple adjacent lines tend to affect the same symbol position in each affected codeword in the line. Thus, Multi-ECC lets multiple lines with the same column address in adjacent rows share the same set of checksums to reduce checksum overhead.

To share checksums across multiple lines, we let every codeword belong to a column checksum group. A column checksum group is comprised of a set of codewords in the same column as well as a set of checksums computed from these codewords. Figure 4 illustrates a column checksum group; the figure shows only two codewords per checksum group for clarity of illustration. In practice, we assign 256 codewords per checksum group to reduce the storage overhead of the checksums to $1/256 \approx 0.4\%$. Figure 4 shows that each checksum is computed using the same data symbol in

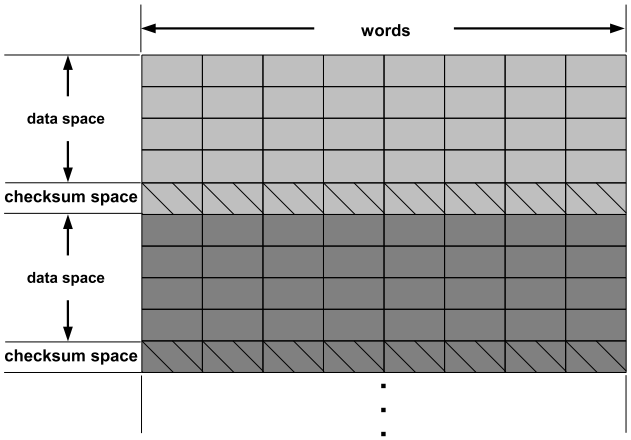

Figure 5: The data and checksum regions in a bank; in the example, five adjacent codewords in the same column belong to the same column checksum group, with the fifth being the checksum of the column checksum group.

different codewords and that each checksum and the data it protects are stored in the same device. When a fault occurs in a device, the checksum stored in the faulty device no longer equals the checksum newly computed from the data symbols; the device that stores the mismatching checksum is thereby identified as the faulty device.

To store checksums in memory, we use fixed/reserved physical pages in memory. Specifically, Multi-ECC stores the checksums in memory rows adjacent to the checksum groups they protect. Figure 5 shows the codewords in a bank and their checksums. In the figure, each grid represents a codeword. All the grids in the same column with the same color belong to the same column checksum group. The figure shows an example scenario in which there are only four codewords (instead of 256) per checksum group.

The following steps are needed to use the checksums to correct errors. During memory operations, when errors are detected in the received codeword by the RS check symbols, all codewords in the corresponding column checksum group are read out. This allows nine new checksums, one corresponding to each device in the rank, to be computed for the checksum group, which are to be compared against the corresponding nine checksums stored in memory. If there is exactly one newly computed checksum that does not match its counterpart in memory, the bad symbol is successfully localized; the localized bad symbol then can be corrected via erasure correction using the RS check symbol. Otherwise, the OS is alerted that a DUE has occurred.

4.4 Impact on DUE Rate

When the checksums are shared across a checksum group, the checksums can correct all errors that are due to a single faulty device in a rank of devices. Correcting all errors due to a single faulty device is one of the primary reliability targets of chipkill correct (the other is to detect all errors due to two faulty devices in a rank). However, because many prior chipkill-correct solutions can guarantee correction of all single-symbol errors, they can also correct errors due to some combinations of faults in two different devices in a rank if the faults in those two devices do not affect any

common codeword. The downside of sharing the checksums across a checksum group is that it reduces the correction coverage of this type of errors; when checksums are shared, errors due to two faults in two different devices in a rank can be corrected only if the two faults do not affect a common checksum group, instead of only a common codeword. This is because two faults in two different devices affecting the same checksum groups will cause two error locations to be reported, which cannot be corrected by Multi-ECC because each codeword contains only one RS check symbol.

The occurrence of single-symbol errors that are correctable by prior chipkill-correct solutions but not by Multi-ECC is rare, however. This requires two devices develop a row fault or symbol fault within 257 rows of each other (because each checksum group spans 256 rows in memory storing data plus one row in memory storing checksums). If two faults in two different devices affecting a common checksum group were larger DRAM faults – such as column, bank, or device faults that span orders of magnitude more codewords than there are codewords in a single checksum group – these two faults most likely also will affect a common codeword to result in a double-symbol error, which is also uncorrectable by prior chipkill-correct solutions.

The probability of developing these fault combinations is small, however. We estimate this probability to be roughly equal to the probability of a 9-device rank developing two or more symbol faults and/or row faults in two or more devices that are within 257 rows of each other by the end of the typical server lifespan of 7 years [20]. Note that this is a conservative estimate; this estimate regards two symbol faults in two adjacent rows as uncorrectable, while in reality these two symbol faults have to be located within the same column of words as well. For our calculation, we used the symbol fault and row fault incidence rates reported in [23], and the same DIMM size that was studied in [23] (i.e., 18 devices per DIMM). The calculation shows that the rate of uncorrectable single-symbol error is less than $3.17 \cdot 10^{-7}$ per 10 years per DIMM. To put this in perspective, a commercial DUE rate target for servers is once per 10 years per server [7]. Even if a server were to consist of 1000 DIMMs, the incidence rate of uncorrectable single-symbol error in Multi-ECC is still only 0.03% of the DUE rate target.

4.5 Extending Multi-ECC for Double Chipkill Correct

Some commercial chipkill-correct solutions, such as Double Chip Sparing [11, 3], provide double-chipkill correct, which can correct double-symbol errors. Extending Multi-ECC to provide double-chipkill correct is straightforward. Having two check symbol per codeword, instead of one, provides error correction of up to two bad symbols per codeword. In addition to providing error correction for double-symbol errors, having two check symbols per codeword allows double-chipkill correct Multi-ECC to guarantee detection of all double-symbol errors and to provide high detection coverage of triple-symbol errors.

To implement double-chipkill correct Multi-ECC at the same storage overhead as our primary Multi-ECC implementation with nine devices per rank, one needs to also double the number of data symbols per codeword, and therefore, double the total number of devices per rank. As such, double-chipkill correct Multi-ECC requires 16 data devices and two redundant devices per rank, which results in an overall rank size of 18 devices.

5. IMPLEMENTATION DETAILS

This section describes the important details necessary for the physical implementation of Multi-ECC. First, we describe the changes necessary to implement RS codes with 16-bit symbols because commercial chipkill-correct solutions use only RS codes with 4- and 8-bit symbols [26, 4]. Second, we describe checksum operation in Multi-ECC because commercial solutions do not use any checksums. Finally, because the error localization required for error correction in Multi-ECC requires accessing every line in a checksum group, error correction in Multi-ECC can be slower than that of commercial chipkill-correct solutions; we present optimizations to allow Multi-ECC to perform error correction as fast as commercial chipkill-correct solutions for permanent/recurring faults. None of these modifications requires changing the memory processor interface, DRAM devices, or memory modules.

5.1 Encoding and Decoding 16-Bit RS Codes

Multi-ECC uses a 16-bit RS code to provide high error-detection coverage of double-symbol errors with a single check symbol per codeword (see Section 4.1). This subsection evaluates the area and latency overheads of the encoder and decoder circuits when using a 16-bit RS code.

The area overhead of RS encoders is $m^2 \cdot k \cdot r$, where m is the symbol width in bits and k and r are the number of data and redundant symbols per codeword [18]. As such, despite its larger symbol width, Multi-ECC requires a smaller or same-sized RS encoder compared to commercial chipkill correct due to the smaller number of data and redundant symbols per codeword. Meanwhile, the latency overhead of RS encoders is $log_2(m \cdot k)$ [18]. Again, the RS encoder of Multi-ECC has lower or similar latency than that of commercial chipkill correct due to the smaller number of data symbols per codeword.

Because Multi-ECC uses the RS check symbols for error detection and erasure correction, the decoder consists of two parts: the error-detection circuit and the erasure-correction circuit. Error detection can be performed by recomputing the check symbol of the received data symbols to see if it matches the received check symbol stored in memory. As such, the error-detection circuit is the same as the encoder plus a comparator.

The primary overhead of having 16-bit symbols is in the erasure-correction circuit. This is because error correction requires performing finite field multiplications and division, which can be implemented via a precomputed log and an anti-log table, each containing 2^n fields with n bits per field, where n is the number of bits per symbol [9]. For RS codes with 16 symbols, this translates to two tables with 128KB per table. Because these tables hold precomputed constants, they can be implemented in the memory controller as a ROM. Because each ROM cell consists of one transistor, instead of six transistors in a cache cell, we estimate that the area of each ROM table is one-sixth that of an equivalent direct-mapped cache with 16-bit output size. Using Cacti [2] to calculate cache area, we estimate that the combined area of the two ROM tables is only 0.7% that of a 16-way set associative 2MB SRAM cache.

5.2 Updating the Checksums

Unlike the RS check symbols which are stored in redundant devices, checksums are stored in reserved physical pages in memory. As a result, updating the checksums requires additional memory accesses. The appropriate checksums need to be updated whenever the value of a line in memory is modified during writeback to memory. Note that proposals such as VECC and LOT-ECC also require additional memory accesses to update their error-correction bits.

To reduce the number of additional memory accesses to update the checksums, we use a similar optimization technique as used in VECC, which is to cache the checksums in the processor. We propose providing the memory controller with a private cache to store the checksums, which we will refer to as the checksum cache. To avoid increasing the overall area of the processor, we propose subtracting one way from the last-level cache and using its area to implement a cache for the checksums in the memory controller.

Updating a cacheline of checksums in the checksum cache when a dirty cacheline has to be written back to memory is a two-step procedure. Step 1 is to subtract from the checksum cacheline the effect of the old/clean value of the data cacheline on the checksum cacheline. Step 2 is to add to the checksum cacheline the effect of the current/dirty value of the data cacheline on the checksum cacheline. The effect of a data cacheline on a checksum cacheline is the contribution of the values in the data cacheline on the values of the checksums in the checksum cacheline, which is a function of the exact code selected to implement the checksums.

To implement Step 1, we propose sending the value of a clean data cacheline to the memory controller whenever the clean data cacheline is modified for the first time (e.g., when its dirty bit is being set). If the corresponding checksum cacheline is not found in the checksum cache, the memory controller first needs to retrieve the checksum from memory. When the checksum cacheline has been received, the memory controller can then modify the checksum cacheline using the clean value of the data cacheline to complete Step 1. Meanwhile, when a dirty cacheline is evicted/written back to memory, the memory controller has to modify the corresponding checksum cacheline using the dirty value of the data cacheline to complete Step 2.

This procedure for updating checksum has implications for error localization because the checksums are not updated by the current value of the dirty cachelines until the lines are written back to memory. One method to account for this simply is to update the checksum cacheline by searching for dirty cachelines that belong to the checksum group with errors and then writing them back to memory before performing error localization. Instead of updating the checksum cacheline, a second method is to modify the error-localization procedure such that dirty cachelines do not participate in the computation of the new checksums to be compared against the checksum cacheline.

Finally, checksum caching also has implications for the placement of virtual pages in memory. To maximize the spatial locality of the checksum lines in the checksum cache, the OS ideally should allocate adjacent virtual pages to physical pages in adjacent rows in the same bank of memory because the lines across these rows share the same checksums. In this way, when the checksum cacheline for a single virtual page has been fetched into the checksum cache, memory writebacks to the neighboring 255 virtual pages will not suffer any checksum misses in the checksum cache. On the other hand, storing adjacent virtual pages in different banks can reduce the number of expensive bank conflicts during a sequence of memory accesses to these adjacent virtual pages [14]. Because a thread typically does not benefit from interleaving adjacent virtual pages over more than 16 banks [14], we propose interleaving adjacent virtual pages among eight different banks to balance between a high checksum cache hit rate for the checksum cachelines and a low bank conflict rate for memory accesses requested by the applications.

5.3 Optimizing Error Correction for Permanent Faults

Error correction in Multi-ECC may incur high overhead, because it requires reading out all the lines in the checksum group to perform error localization. This is expensive because it requires 256 additional memory accesses to perform error correction compared to commercial solutions. This performance degradation due to error correction is particularly pronounced for permanent device-level faults, such as the permanent complete device fault and the permanent bank fault, which result in repeated and frequent errors that require correction. Because these faults only affect performance, not correctness (i.e., Multi-ECC can correct errors due to these faults), it may be possible in certain scenarios simply to replace the DIMMs with these faults during a prescheduled downtime. The remainder of this section explores ways to reduce the performance impact of these faults when DIMM replacement may be infeasible.

To reduce the performance overhead of error correction for permanent faults, we observe that if only one codeword in the checksum group contains a bad symbol (e.g., due to a symbol fault or a row fault), one can retire the page containing the bad symbol to prevent future expensive accesses to the same bad codeword. However, if multiple codewords in the checksum group contain errors (e.g., when a device develops a serious device-level fault such as a bank fault), simply retiring/disabling all pages with errors can significantly reduce the total usable memory capacity.

For a checksum group with multiple codewords that are erroneous in the same symbol position, we propose remapping the bad symbols in the faulty device responsible for the errors to another device. The remapping of bad symbols is a technique that is used currently by double-chip sparing [11, 10]. In Multi-ECC, remapping the bad symbols in a faulty device to another device is a two-step procedure. Step 1 is to find a device to remap the symbols to. Step 2 is to record which symbol position in the faulty checksum group has been remapped.

Implementing Step 1 of the remapping process is straightforward for double-chipkill correct Multi-ECC because it allocates two check symbols per codeword; one of the two check symbols can instead become the destination of the remapping while the second check symbol can still provide erasure correction for a second bad symbol. However, this is more complicated in our primary Multi-ECC with only nine devices per rank, because it allocates a single check symbol per codeword. Symbol remapping in our primary Multi-ECC can be achieved with the help of a technique proposed in ARCC [12]. When faults are detected, ARCC combines two adjacent lines in two different memory channels into a single large line so each codeword in the large line can be striped across twice as many DRAM devices. This in turn allows each codeword to contain twice as many

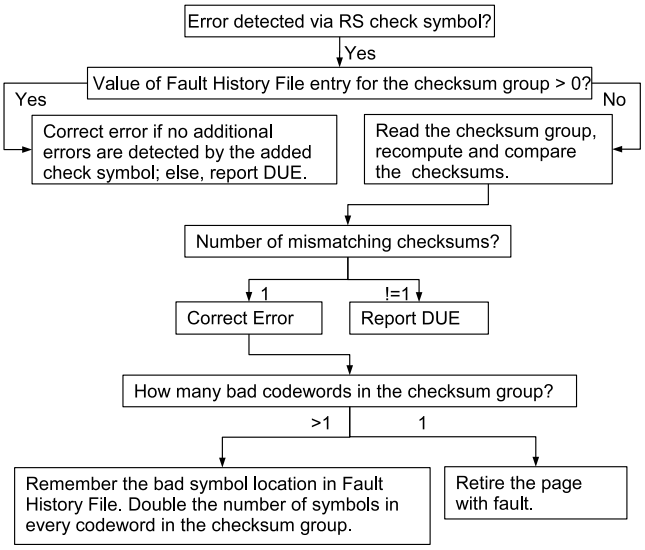
Error detected via RS check symbol?
Yes
Value of Fault History File entry for the checksum group > 0?
Yes
No
Correct error if no additional errors are detected by the added check symbol; else, report DUE.
Read the checksum group, recompute and compare the checksums.
Number of mismatching checksums?
1
!=1
Correct Error
Report DUE
How many bad codewords in the checksum group?
>1
1
Remember the bad symbol location in Fault History File. Double the number of symbols in every codeword in the checksum group.
Retire the page with fault.

Figure 6: Error correction optimized for correcting errors due to permanent faults.

check symbols without affecting the storage overhead [12]. Multi-ECC can rely on the same technique to double the number of check symbols per codeword so the bad symbol can be remapped to the newly added check symbol, while error detection of additional symbol errors can be provided by the other check symbol. Although Multi-ECC uses the same technique as ARCC to increase the number of check symbols per codeword when faults are detected, their reasons for doing so are completely different. ARCC does so to increase the strength of the chipkill-correct solutions (e.g., to be able to detect double-symbol errors), while Multi-ECC does so to reduce the performance overhead of error correction (i.e., not for reliability because Multi-ECC can already detect double-symbol errors).

To implement Step 2 (remembering which symbol position in the checksum group has been remapped), we use the Fault History File to store which device contains the bad symbols in the checksum group so error correction for future memory accesses to the same column checksum group do not require the expensive fault-localization step. There is a fixed entry in the Fault History File for every checksum group so the memory controller can easily find the desired entry given the address of a data line. These entries are stored in reserved physical pages in memory. The initial value of every entry is 0; after bad symbols in a checksum group have been localized to a particular device, the position of the faulty device is recorded in the entry corresponding to the group. Each entry takes a 5-bit value between 0 and 18, where 18 is the total number of symbols per codeword after two regular nine-symbol codewords have been joined to form a single large codeword. To protect the entries against memory faults, we store three copies of the same entry in three different devices to provide triple modular redundancy. This translates to an overall overhead of only 15 bits for each group of 256 codewords, which translates to an overall storage overhead of $15/(256 \cdot 18 \cdot 16) = 0.02\%$.

With the help of the Fault History File, the expensive fault localization step is required only when a new fault develops. Since the incidence rate of new faults is very low (e.g., our calculation using the measured DRAM fault rate reported in [23] shows that a fault occurs once per two years per 100GB of memory), the overall performance overhead of fault localization, and hence error correction, is negligible.

Finally, similar to the checksums, we propose caching the entries of the Fault History File in the processor so that they do not always have to be read from memory. Because only a single 5-bit value needs to be cached for every group of data lines, a whole 64B cacheline of entries has extremely high coverage of data lines (e.g., a cacheline of 5-bit fault history entries covers more than 6,500 data lines). As such, they can be cached effectively in a much smaller cache than the checksum cache.

Figure 6 summarizes the optimized error-correction procedure described in this section while Table 1 compares the worst case and the average cases of correction overhead of Multi-ECC, in terms of the number of additional accesses to memory per application access to memory, to that of other chipkill-correct solutions.

Table 1: Error-correction overhead

	Worst Case	Average Case
Commercial	0	0
VECC	1	[0,1][2]
LOT-ECC	1	~1
Multi-ECC	256	~0

6. METHODOLOGY

In our evaluation, we compare the power and performance of Multi-ECC against commercial chipkill correct, VECC, and LOT-ECC in a fault-free memory system. We simulate a quadcore processor using M5 [6]; the parameters for the simulated processor are provided in Table 2. The processor contains a 2MB 16-way set-associative last-level cache. To investigate the memory traffic overhead due to updating the checksums, we take one way out of the 16-way set associative last-level cache to model a 128KB 32-way set-associative checksum cache in the memory controller. Because both Multi-ECC and VECC cache the error-correction resources, we also cache the error correction resources of LOT-ECC to perform a fair comparison.

We simulate the memory system using DRAMsim [19]. Similar to the evaluation of VECC, we choose the open-page row buffer policy. We choose SDRAM High Performance Map in DRAMsim [19] as the physical address-mapping policy. All memory configurations use DDR3 and take up 144 data pins in the memory bus. The DDR3 devices operate at 1.3 GHz; their timing and power characteristics are extracted from Micron datasheets [17]. We choose the common memory output granularity of 64B for all memory configurations. To accomplish this for commercial chipkill correct baseline with 36 devices per rank, we rely on the burst chop option in DDR3, which reduces the burst length from the usual eight bursts down to four bursts [17]. We optimistically model burst chop such that consecutive four burst accesses can always take place back to back. In practice, however, burst chop requires four useless memory cycles during which only waste energy and occupy bandwidth between various memory access combinations.

[2]The average case correction overhead of VECC depends heavily on cache size and memory access patterns.

Table 2: Processor microarchitecture

Clock	SS Width	L1 D$, I$	L1 lat.	L1 Assoc
2 GHz	2	32 kB	1	1
L2$	L2 Assoc	L2 lat.	L2 MSHR	line Size
2MB	16	14	512	64B

Table 3: Memory configurations

ECC	Channel	I/O Pin	Ranks/ Chan	Chips/ Rank	Mem. Size
Commercial	1	X4	1	36	16GB
VECC	2	X4	1	18	15GB
LOT-ECC	2	X8	2	9	14GB
Multi-ECC	2	X8	2	9	16GB

Table 3 shows the memory configurations of commercial chipkill correct, VECC, LOT-ECC, and Multi-ECC used in our evaluation. The last column of Table 3 shows the effective memory capacities of each memory configuration after excluding the storage overheads. The column shows that VECC and LOT-ECC have 1GB and 2GB less effective memory capacity, respectively, than commercial chipkill correct and Multi-ECC even though the total number of DRAM devices is the same in each configuration. Our decision to evaluate design points for an equal amount of total physical memory capacity instead of an equal amount of usable memory capacity is consistent with the methodology used in VECC and LOT-ECC.

We use both multi-threaded PARSEC [5] workloads and multi-programmed SPEC workloads. The compositions of the multi-programmed SPEC workload are listed in Table 4. To simulate a multi-programmed SPEC workload, we fast-forward each benchmark in the workload by 1 billion instructions and evaluate the workload for 1 billion cycles. For the PARSEC workloads, which have to be simulated in the FULL SYSTEM mode [8], we evaluate each benchmark for 1 billion cycles after the OS boots.

7. RESULTS

7.1 Power and Performance

Figure 7 shows the power of Multi-ECC normalized to commercial chipkill correct, VECC, and LOT-ECC. The figure shows that the power reduction compared to commercial chipkill correct correlates closely with the rate of memory access, which is given in Figure 8. This is because the key advantage of Multi-ECC relative to commercial chipkill correct is having a smaller rank size (nine devices instead of 36 devices); therefore, the higher the memory-access rate, the greater the power savings. Compared to commercial chipkill correct, Multi-ECC achieves maximum and mean power reductions of 38% and 27%, respectively. Compared to VECC, a similar trend of power reduction exists. The mean power reduction with respect to VECC is 23%.

Compared to LOT-ECC, Figure 7 shows that the mean difference in memory power consumption is small (e.g., only

Table 4: Simulated SPEC workloads

Mix1	mesa;swim;apsi;sphinx3
Mix2	lucas;gromacs;swim;fma3d
Mix3	sjeng;swim;facerec;ammp
Mix4	mcf_2006;libquantum;omnetpp;astar
Mix5	calculix;swim;art110;omnetpp
Mix6	lbm;facerec;h264ref_foreman_base;ammp

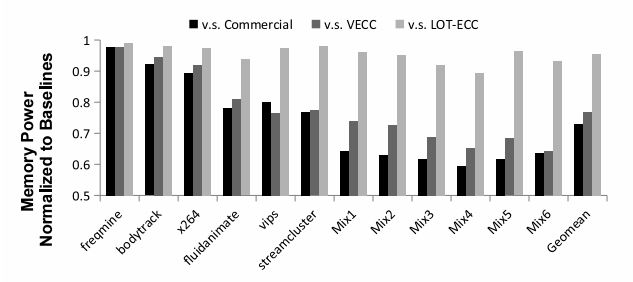

Figure 7: Power of Multi-ECC versus the baselines.

Table 5: Storage overhead

	Commercial	VECC	LOT-ECC	Multi-ECC
Storage Overhead	12.5%	21.9%	26.5%	12.9%

4%). This is because Multi-ECC and LOT-ECC have the same number of devices per rank. However, there are two key advantages of Multi-ECC compared to LOT-ECC in terms of storage overhead and error-detection strength. Recall from Section 2 that LOT-ECC requires a storage overhead of 26.5%, instead of the conventional 12.5%. Table 5 summarizes the storage overhead of Multi-ECC as well as the various baselines. Due to its high storage overhead, the evaluated LOT-ECC memory system contains 2GB less usable memory than commercial chipkill correct and Multi-ECC (see Table 3). To provide the same usable memory capacity, LOT-ECC requires 12.5% additional physical memory, which would result in a minimum of 12.5% increase in background power consumption. When we take this increase in background power into account for a comparison with equal usable memory capacity, Multi-ECC reduces power by 11.6% compared to LOT-ECC. Recall from Section 2 that LOT-ECC can significantly reduce reliability compared to other chipkill-correct solutions because even a single-symbol error can cause SDC under LOT-ECC. (For more detailed error detection/correction comparison, see Section 7.2.)

Figure 9 shows the IPC of Multi-ECC normalized to the baselines. The figure shows that Multi-ECC improves IPC by a mean of 3% compared to commercial chipkill correct. This slight improvement in performance is due to the increase in rank-level parallelism because Multi-ECC uses more ranks per channel to provide the same total memory capacity as commercial chipkill correct (see Table 3). For the same reason, Multi-ECC improves IPC by a mean of 1% com-

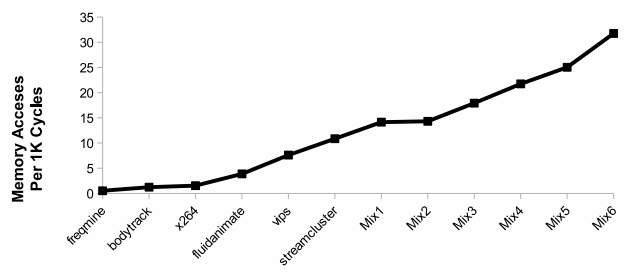

Figure 8: Number of application memory accesses per 1000 CPU cycles.

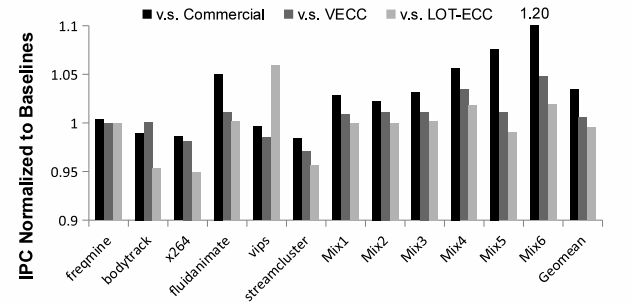

Figure 9: IPC of Multi-ECC versus baselines.

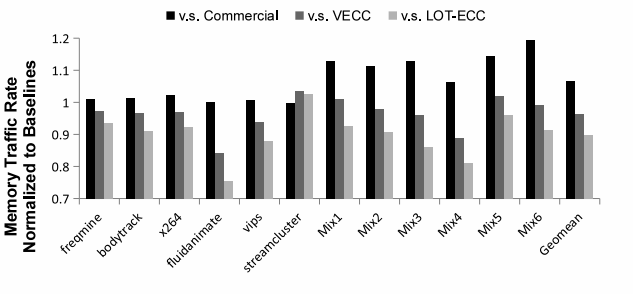

Figure 10: Memory traffic versus baselines.

pared to VECC. However, the IPC of Multi-ECC is 0.5% lower than that of LOT-ECC. This is because Multi-ECC interleaves adjacent virtual pages across only eight different banks (see Section 5.2), while LOT-ECC has no such restriction. This increases the number of expensive long-latency bank conflicts compared to LOT-ECC, resulting in slightly lower performance. When compared to the commercial baseline and VECC, the slight reduction in performance due to the increased bank conflicts is hidden by the increase in performance due to the increase in rank-level parallelism.

Finally, to investigate the effectiveness of the checksum cache, we compared the memory access rate of Multi-ECC against the memory-access rates to the various baselines. Figure 10 shows the memory-access rate of Multi-ECC normalized to the baselines. The figure shows that Multi-ECC increases memory traffic by a mean of 7% compared to the idealized commercial baseline, which does not require any additional memory accesses to update its error-correction resources. On the other hand, the figure shows that the memory traffic rate of Multi-ECC is 4% and 10% lower than that of VECC and LOT-ECC, respectively. This is because while each checksum cacheline in Multi-ECC covers 256 data lines, the corresponding error-correction cachelines in VECC and LOT-ECC cover only 16 and eight data lines, respectively.

7.2 Reliability

Table 6 summarizes the combined rates of undetectable single-symbol and double-symbol errors in Multi-ECC and in the various baselines. The undetectable-error rate in Table 6 for Multi-ECC is taken from Section 4.2. To calculate the undetectable-error rate of LOT-ECC, which cannot detect any address-decoder faults even if only a single device in a rank is affected by an address-decoder fault and all other devices are error-free (see Section 2), we rely on

Table 6: Combined rate of undetectable single- and double-symbol errors.

	Detects all single device errors?	Combined Rate of undetectable single and double symbol errors
Commercial	yes	0
VECC	yes	0
LOT-ECC	no	>0.3 per 1000 yrs *per DIMM*
Multi-ECC 1 symbol/device	yes	<0.0015 per 1000 yrs *per system*
Multi-ECC 2 symbols/device	yes	$< 1.5 \cdot 10^{-8}$ per 1000 yrs *per system*

the DRAM fault rates reported in a recent large-scale field study of DRAM faults performed by Siddiqua et al. [22]. The study attributes 17% of soft faults to block faults, a type of address-decoder fault, and reports that 17% of all memory systems with faults suffered from soft faults; using this report, we conservatively estimate that block faults account for roughly 1% of all DRAM faults. Because other field studies of DRAM faults have shown that 3% [23] to 8% [21] of all DIMMs have been reported to develop faults during each year, we calculate that the SDC rate of LOT-ECC due to block faults affecting a single device is approximately 0.3 SDC to 0.8 SDC per 1,000 years per DIMM. To put this in perspective, a commercial SDC rate target for servers is one SDC per 1,000 years per entire server system, not just for the memory subsystem [7].

Meanwhile, our calculation in Section 4.4 shows that the uncorrectable-error rate due to single-symbol errors for Multi-ECC is less than $3.17 \cdot 10^{-7}$ per 10 years per DIMM, which results in only a 0.03% increase to a commercial target DUE rate of once per 10 years per system [7] even if the system contains 1000 DIMMs (servers typically contain many fewer). This is four orders of magnitude lower than that of the undetectable-error rate of LOT-ECC given in Table 6. This is not surprising considering that while a single fault affecting a single device can result in an undetectable error in LOT-ECC, uncorrectable single-symbol errors in Multi-ECC are caused by two or more symbol faults or row faults in two different devices in a rank in close spatial proximity of each other (See Section 4.4).

8. RELATED WORK IN OTHER AREAS

Multi-ECC uses RS codes in one dimension to perform error detection and erasure correction, and checksums in a second dimension to provide error localization. As such, it is related to other works that use two-dimensional coding. 2D coding is used by Manoochehri et al. [16] to reduce the storage overhead of error correction in the cache, Kim et al. [13] to correct multi-bit cluster errors in embedded cache, and Longwell [15] to reduce the latency of error correction for on-chip memory accesses. To the best of our knowledge, Multi-ECC is the first to employ two-dimensional coding to enable linear block codes with a single check symbol per codeword to provide chipkill correct. Multi-ECC is also the first to use two-dimensional coding to exploit the observation that a single faulty DRAM device in a rank only corrupts the same symbol of the codewords to reduce the storage overhead of error correction in chipkill-correct memory systems with minimal impact on reliability.

Another work related to Multi-ECC is redundant array

of independent disks (RAID), commonly used to provide error correction in hard-drives. RAID uses checksums stored within each data disk to provide error detection and localization, and RS check symbols stored in a redundant disks to provide erasure correction. There are two key differences between RAID and Multi-ECC. First, RAID does not reuse the same RS check symbol intended for erasure correction for error detection. Therefore, it does not exploit the strong detection capability of RS check symbols (i.e., the ability of a single RS check symbol to guarantee detection of single-symbol errors and to provide high detection coverage of double-symbol errors), as does Multi-ECC. Instead, by using checksums for error detection, it is unable to detect errors due to address-decoder faults in a single device, as does LOT-ECC. Second, RAID does not share checksums across multiple 'lines', as does Multi-ECC; because an output line of a hard drive is often 4KB, the storage overhead of the checksums is very small even though each 4KB line has its own dedicated checksum.

9. CONCLUSION

As the memory capacity of servers increases, memory power consumption and reliability become increasingly important concerns. Chipkill correct has been commonly used in servers to provide the desired memory reliability. However, existing chipkill-correct solutions incur high memory power and/or storage overheads.

In this paper, we observe that the high overhead of prior chipkill-correct solutions is due to requiring a large number of check symbols per codeword (i.e., a minimum of three check symbols per codeword). The root cause of the large number of check symbols per codeword is in turn due to using a dedicated correction-check symbol per codeword for error correction. Multi-ECC reduces the number of check symbols per codeword to one by exploiting two observations. First, because DRAM faults often affect the same symbol of codewords in memory, it suffices to allocate a common set of error-correction resources to a group of codewords for error correction instead of allocating a dedicated correction-check symbol to each codeword. Second, we observe that using a dedicated check symbol per codeword for error correction, as does prior chipkill-correct solutions, significantly reduces the effectiveness of error detection, and, therefore, propose an alternate method for error correction.

Instead of relying on a dedicated check symbol per codeword for error correction, Multi-ECC relies on erasure correction. Multi-ECC uses a single check symbol per codeword to provide both erasure correction and error detection. Meanwhile, Multi-ECC keeps the overhead of erasure correction low by sharing error-localization checksums, required for erasure correction, across multiple lines in memory. Overall, Multi-ECC incurs low dynamic memory access power (e.g., only nine devices per rank) at low storage overhead (12.9%), and with only minimal impact on reliability. Our evaluation using workloads across a wide range of memory access rates shows that Multi-ECC reduces memory power consumption by a mean of 27% and up to 38% compared to commercial chipkill-correct solutions.

10. ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their feedback. The work was partly supported by a gift from Oracle.

11. REFERENCES

[1] "AMD, BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors," 2007. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf

[2] "CACTI 5.3," 2008. [Online]. Available: http://quid.hpl.hp.com:9081/cacti

[3] "Overview of the IBM Blue Gene/P project," *IBM J. Res. Dev.*, vol. 52, no. 1/2, pp. 199–220, Jan. 2008.

[4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future Scaling of Processor-memory Interfaces," *SC*, pp. 42:1–42:12, 2009.

[5] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton, NJ, USA, 2011.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *MICRO*, 2006.

[7] D. C. Bossen, "CMOS Soft Errors and Server Design," *IEEE Reliability Physics Tutorial Notes*, 2002.

[8] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler, "Running PARSEC 2.1 on M5."

[9] W. A. Geisel, *Tutorial on Reed-Solomon Error Correction Coding.* National Aeronautics and Space Administration, Lyndon B. Johnson Space Center, 1990.

[10] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," *SIGARCH Comput. Archit. News*, pp. 111–122, 2012.

[11] Intel, "RAS Features of the Mission-Critical Converged Infrastructure," 2010.

[12] X. Jian and R. Kumar, "Adaptive Reliability Chipkill Correct," *HPCA*, pp. 270 – 281, 2013.

[13] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding," *MICRO*, pp. 197–209, 2007.

[14] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," *PACT*, pp. 367–376, 2012.

[15] M. L. Longwell, "Method and apparatus for error detection and correction," 2010, US Patent number: 7644348.

[16] M. Manoochehri, M. Annavaram, and M. Dubois, "CPPC: Correctable Parity Protected Cache," *ISCA*, pp. 223–234, 2011.

[17] MICRON, "2Gb: x4, x8, x16 DDR3 SDRAM," *MICRON*.

[18] S. Morioka, "Design Methodology for a One-shot Reed-Solomon Encoder and Decoder," *ICCD*, pp. 60–67, 1999.

[19] U. of Maryland, *University of Maryland Memory System Simulator Manual.*

[20] M. K. Qureshi, "Pay-As-You-Go: Low-overhead Hard-error Correction for Phase Change Memories," *MICRO*, pp. 318–328, 2011.

[21] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: a Large-scale Field Study," *SIGMETRICS*, pp. 193–204, 2009.

[22] T. Siddiqua, A. E. Papathanasiou, A. Biswas, and S. Gurumurthi, "Analysis and Modeling of Memory Errors from Large-scale Field Data Collection," *SELSE*, 2013.

[23] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," *SC*, 2012.

[24] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems," *ISCA*, pp. 285 – 296, 2012.

[25] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," *ISCA*, pp. 175–186, 2010.

[26] D. H. Yoon and M. Erez, "Virtualized ECC: Flexible Reliability in Main Memory," *MICRO*, pp. 285 – 296, 2010.